Un système de modules avancé pour SIGNAL

David Nowak, Jean-Pierre Talpin, Thierry Gautier

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Un système de modules avancé pour SIGNAL

David Nowak, Jean-Pierre Talpin, Thierry Gautier

N° 3176

Juin 1997

THÈME 1

ISSN 0249-6399

Un système de modules avancé pour SIGNAL

David Nowak, Jean-Pierre Talpin, Thierry Gautier

Thème 1 — Réseaux et systèmes
Projet EP-ATR

Rapport de recherche n° 3176 — Juin 1997 — 34 pages

Résumé : Nous proposons un système de modules avancé pour SIGNAL permettant de définir des unités génériques, des types abstraits et de paramétrer les modules par d'autres modules. La première tâche a été de formaliser le typage de SIGNAL sous forme de règles d'inférences, puis d'en déduire un algorithme de synthèse automatique des types.

Mots-clé : modules, typage, SIGNAL, temps-réel

(Abstract: pto)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 02 99 84 71 00 – Télécopie : (33) 02 99 84 71 71

An advanced modules system for SIGNAL

Abstract: We propose a module system for SIGNAL. It allows to define generic units, abstract types and to parametrize modules by other modules. Our first work was to formalize the type system of SIGNAL with inference rule and then to deduce a type inference algorithm.

Key-words: modules, type system, SIGNAL, real-time

Introduction

Un système réactif est un système qui interagit avec son environnement : Il en reçoit des informations et agit sur lui. Souvent, une défaillance d'un système réactif peut entraîner des pertes matérielles ou humaines. Le système doit pouvoir suivre le rythme de l'environnement et être capable de réagir instantanément à tous stimuli (i.e. réagir dans un temps négligeable).

Traditionnellement, ces systèmes ont été programmés en C, en ADA, ou en OCCAM. ces langages sont tous des langages impératifs asynchrones qui ne garantissent pas la réactivité du logiciel. Le langage SIGNAL est développé pour pallier à ces difficultés.

Le typage statique a pour but de garantir l'exécution correcte des programmes bien typés. Un programme bien typé ne s'arrêtera pas sur une erreur de type. De plus, le typage peut aider à l'optimisation du code compilé.

Dès qu'un programme devient important en taille, il est nécessaire de le structurer de manière à, premièrement, pouvoir en modifier une partie sans avoir à comprendre tout le reste du programme et, deuxièmement, pouvoir le recompiler en utilisant un minimum de ressources de mémoire et de temps. À ces fins, on découpe les programmes en modules, qui sont des morceaux de code pouvant être *compris* séparément, et en unités de compilation, qui sont des morceaux de code pouvant être *compilés* séparément. Bien que présentant des similitudes, ces deux concepts ne peuvent en général pas être identifiés. Certes, ils ont en commun des interfaces où sont spécifiés les types et les objets exportés, mais lorsqu'ils sont identifiés (comme en Modula-2) l'expressivité des modules est fortement réduite.

Dans le premier chapitre, nous présenterons le langage SIGNAL. Puis, dans le deuxième chapitre, nous formaliserons le système de type de SIGNAL et nous en déduirons un algorihme d'inférence des types. Enfin, dans le troisième chapitre, nous spécifierons un système de modules.

1 Le langage SIGNAL

SIGNAL fait partie de la famille des langages synchrones à flots de données. Il est utilisé pour le développement d'applications temps-réel. Dans certains cas, une erreur de programmation peut entraîner des pertes matérielles ou

humaines. L'échec du vol inaugural d'Ariane 5 en est un exemple. Il apparaît nécessaire de vérifier les programmes temps-réel par le contrôle des types et par le calcul d'horloges.

1.1 Les principaux opérateurs de SIGNAL

Les objets manipulés par SIGNAL [4] sont les signaux. Un *signal* est une suite de valeurs indicées sur $I\!N$ pour décrire un ensemble discret d'instants. À un instant donné, un signal peut-être absent ou présent. S'il est présent, il peut avoir une valeur. L'*horloge* d'un signal est l'ensemble des instants où le signal est présent. Deux signaux sont dits *synchrones* s'ils ont la même horloge. On notera $\widehat{x}$ l'horloge du signal x, $[y]$ l'horloge où le signal booléen y est présent et vrai, $[\neg y]$ l'horloge où le signal booléen y est présent et faux.

SIGNAL se fonde sur l'hypothèse de synchronisme :

simultanéité : On peut déterminer si deux signaux sont *simultanés* ou se produisent à des instants différents.

instantanéité : Les calculs prennent un temps nul.

SIGNAL , dont la syntaxe est présentée en fig. 1, est un langage déclaratif équationnel.

Un programme est une composition d'équations entre signaux. Ces équations s'écrivent sous la forme $x := e$ où e est une expression définie à l'aide des opérateurs de base suivants :

- L'opérateur de *retard* permet d'accéder à la valeur précédente d'un signal. L'expression $x\$1$ est égale à la valeur précédente du signal x.

- L'opérateur d'extraction **when** permet d'extraire un sous-signal. L'expression x **when** y définit un signal qui est présent et est égal à x quand la condition y est présente et vraie.

- L'opérateur de mélange **default** permet de mélanger deux signaux. L'expression x **default** y est égale à x quand x est présent, sinon elle est égale à y.

u, v		valeur	
x, y, z		signaux	
o		identificateur de fonction instantanée	
f		identificateur de processus	
t	::=	**event** | **boolean** | **short** | **integer** | **long** | **real** | **dreal** | **complex** | **dcomplex** | **char** | **string**	types de base
d	::=	**process** $f = (?\ a_1, \cdots, a_n\ !\ a'_1, \cdots, a'_p)\ p$	modèle de processus
p	::=	$x := e \mid (p_1 \mid p_2) \mid p$ **where** $dl \mid p/x_1, \cdots, x_n$	processus
e	::=	$v \mid x \mid e\$1$ **init** $v \mid e$ **when** $e' \mid e$ **default** $e' \mid o(e_1, \cdots, e_n) \mid f(e_1, \cdots, e_n)$	expression
a	::=	$x \mid t\ x$	paramètre formel
dl	::=	**end** $\mid a; dl \mid d; dl$	liste de déclarations

FIG. 1: *syntaxe de* SIGNAL

– Une *fonction instantanée* permet d'effectuer un calcul entre signaux. $f(x_1, \cdots, x_n)$ est égal à $f(v_1, \cdots, v_n)$ où $v_1, \cdots, v_n$ sont les valeurs de $x_1, \cdots, x_n$.

Un processus peut en appeler un autre défini localement comme un sous-processus. Prenons par exemple le processus `PENDULE` (cf. fig. 2) qui utilise le sous-processus `CPT_MOD` (compteur modulo `N`).

Ce processus `PENDULE` renvoie chaque seconde le jour et l'heure à partir d'une source de tops battant la seconde.

1.2 La sémantique dynamique de SIGNAL

Nous décrivons en figure 3 la sémantique opérationnelle de SIGNAL par une relation de transition [2].

La relation de transition, notée $\longrightarrow$, est définie par un ensemble de règles de la forme :

$$\frac{C}{p \xrightarrow{m} p'}$$

```
process PENDULE = (?event TOPS !integer JOUR, HEURE, MINUTE, SECONDE)
  (| (SECONDE, NOUV_MINUTE) := CPT_MOD{60} (TOPS, TOPS)
   | (MINUTE, NOUV_HEURE) := CPT_MOD{60} (TOPS, NOUV_MINUTE)
   | (HEURE, NOUV_JOUR) := CPT_MOD{24} (TOPS, NOUV_HEURE)
   | ZJOUR := JOUR $1 init 1
   | JOUR := ZJOUR+1 when NOUV_JOUR default ZJOUR
   | synchro (JOUR, TOPS)
   |) where
     event NOUV_MINUTE, NOUV_HEURE, NOUV_JOUR;
     integer ZJOUR;
     process CPT_MOD = {integer N}
       (?event TOP_SORTIE, TOP_INCR !integer CPT; event RAZ)
       (| synchro (CPT, TOP_SORTIE default TOP_INCR)
        | ZCPT := CPT $1 init 0
        | CPT := MOD(ZCPT+1,N) when TOP_INCR default ZCPT
        | RAZ := when CPT=0 when TOP_INCR
        |) where
          integer ZCPT;
          function MOD=(?integer I, J !integer M)
     end
end
```

FIG. 2: *le processus* PENDULE

où C est une condition sur p, m et p'; p et p' sont les états (i.e. les processus avec leurs mémorisation des signaux retardés); et m est un événement (i.e. la valeur ou l'absence des signaux au moment de la transition).

Nous ne détaillerons pas les transitions liées à l'événement vide. Il s'agit, par exemple, dans le cas de la fonction instantanée :

$$x := f(y_1, \cdots, y_n) \xrightarrow{x(\perp)\ y_1(\perp)\cdots y_n(\perp)} x := f(y_1, \cdots, y_n)$$

$$x := y\$1 \textbf{ init } u \xrightarrow{x(u)\ y(v)} x := y\$1 \textbf{ init } v$$

$$\begin{array}{lll}
x := y \textbf{ when } z & \xrightarrow{x(\bot)\ y(v)\ z(\bot)} & x := y \textbf{ when } z \\
x := y \textbf{ when } z & \xrightarrow{x(\bot)\ y(\bot)\ z(v)} & x := y \textbf{ when } z \\
x := y \textbf{ when } z & \xrightarrow{x(v)\ y(v)\ z(true)} & x := y \textbf{ when } z \\
x := y \textbf{ when } z & \xrightarrow{x(\bot)\ y(v)\ z(false)} & x := y \textbf{ when } z
\end{array}$$

$$\begin{array}{lll}
x := y \textbf{ default } z & \xrightarrow{x(v)\ y(v)\ z(\bot)} & x := y \textbf{ default } z \\
x := y \textbf{ default } z & \xrightarrow{x(v)\ y(v)\ z(u)} & x := y \textbf{ default } z \\
x := y \textbf{ default } z & \xrightarrow{x(v)\ y(\bot)\ z(v)} & x := y \textbf{ default } z
\end{array}$$

$$y := o(x_1 \cdots x_n) \xrightarrow{y(f(u_1,\cdots,u_n))\ x_1(u_1)\cdots x_n(u_n)} y := f(x_1 \cdots x_n)$$

$$\frac{p \xrightarrow{m} p'' \quad p' \xrightarrow{m'} p''' \quad m \cap m'}{p|p' \xrightarrow{m \cup m'} p''|p'''} \qquad \frac{p \xrightarrow{m\ x_1(v_1)\cdots x_n(v_n)} p'}{p/x_1,\cdots,x_n \xrightarrow{m} p'/x_1,\cdots,x_n}$$

FIG. 3: *Sémantique dynamique de* SIGNAL

- Le retard nécessite une mémoire où est stockée la valeur précédente du signal. $x := y\$1$ **init** u mémorise la valeur v du signal y et renvoie sa valeur précédente u.
- L'équation $x := y$ **when** z signifie que x prend les valeurs de y quand le signal booléen z est présent et vrai. Sinon x n'est pas défini. On remarque qu'il n'y a pas de changement d'état dans ces transitions.
- L'équation $x := y$ **default** z signifie que x prend les valeurs de y quand celui-ci est présent, sinon il prend les valeurs de z. Si aucun des deux n'est présent, alors x n'est pas défini. On remarque qu'il n'y a pas de changement d'état dans ces transitions.
- Dans le cas d'une fonction instantanée f, celle-ci n'est évaluée que si tous les signaux d'entrée sont présents. Il n'y a pas de changement d'état.

– Soit $dom(m)$ le domaine de l'événement m. On notera $m|_D$ la restriction de m au sous-ensemble D de son domaine. On notera $m \cap m'$ pour dire que les signaux communs à m et m' portent les mêmes valeurs.

$$m \cap m' \Leftrightarrow m|_{dom(m)\cap dom(m')} = m'|_{dom(m)\cap dom(m')}$$

1.3 Le calcul d'horloge

A chaque programme SIGNAL est associé un ensemble de relations entre les horloges des signaux et un Graphe de Dépendances Conditionné (GDC) entre signaux. Pour les calculer, il faut d'abord transformer le programme écrit en syntaxe étendue (présentée en figure 1) dans une syntaxe élémentaire où les sous-expressions sont uniquement des signaux ou des valeurs. Cette traduction, notée $p \rightsquigarrow p'$, est décrite en figure 4.

$$\frac{e \rightsquigarrow p/x \quad e' \rightsquigarrow p'/x'}{e \textbf{ when } e' \rightsquigarrow ((y := x \textbf{ when } x'|p|p')/x, x')/y} \qquad \frac{e \rightsquigarrow p/x}{y := e \rightsquigarrow p[y/x]}$$

$$\frac{e \rightsquigarrow p/x \quad e' \rightsquigarrow p'/x'}{e \textbf{ default } e' \rightsquigarrow ((y := x \textbf{ default } x'|p|p')/x, x')/y} \qquad \frac{p \rightsquigarrow p'' \quad p' \rightsquigarrow p'''}{p|p' \rightsquigarrow p''|p'''}$$

$$\frac{(e_i \rightsquigarrow p_i/x_i)_{i=1,\cdots,n}}{y_1, \cdots, y_p := f(e_1, \cdots, e_n) \rightsquigarrow (y_1, \cdots, y_p := f(x_1, \cdots, x_n)|p_1| \cdots |p_n)/x_1, \cdots, x_n}$$

FIG. 4: *Traduction dans la syntaxe élémentaire*

La figure 5 définit formellement ce qu'est un Graphe de Dépendances Conditionné g. Si c et c' sont des expressions d'horloges, $c \wedge c'$ et $c \vee c'$ sont respectivement l'intersection et l'union des instants définis par c et c'. $\hat{x}\backslash\hat{y}$ est l'ensemble des instants où x est présent et y est absent. $fv(g)$ et $bv(g)$ sont respectivement l'ensemble des variables libres de g et l'ensemble des variables liées de g.

Nous présentons le calcul d'horloge sous la forme d'un système de d'inférence (cf. fig. 6) où G est un environnement qui associe des noms de processus f à leur spécification $(?\ \vec{x}\ !\ \vec{y})\ g$. Les dépendances sont étiquetées par l'horloge

$$\begin{array}{lll} c & ::= & \hat{x} \mid \hat{x}\backslash\hat{y} \mid [x] \mid c \wedge c \mid c \vee c & \textit{horloge} \\ g & ::= & \emptyset \mid g \cup \{\hat{x} = c\} \mid g \cup \{x \xrightarrow{c} y\} \mid \forall x.g \mid g \cup g' & \textit{GDC} \end{array}$$

$$g_x = \{y \xrightarrow{c} x\} \quad g^x = \{x \xrightarrow{c} y\}$$
$$g|_x = g\backslash g_x\backslash g^x \cup (\bigcup_{(y \xrightarrow{c} x, x \xrightarrow{c'} z) \in g_x \times g^x} \{y \xrightarrow{c \wedge c'} z\})$$

$$\forall x.g = \forall x.(g|_x) \quad \forall x.g = g \text{ ssi } x \notin fg(g) \quad \forall x.(g \cup \{\hat{x} = \hat{y}\}) = g[\hat{y}/\hat{x}]$$
$$\forall y.(\forall x.g) = \forall x.(\forall y.g) \quad \forall y.g = \forall x.g[x/y] \text{ ssi } x \notin fv(g) \cup bv(g)$$
$$(\forall x.g') \cup g = \forall x.(g \cup g') \text{ ssi } x \notin fv(g)$$

FIG. 5: *Le graphe de dépendances conditionné*

à laquelle elles sont valables. Une dépendance conditionnelle $x \xrightarrow{c} y$ est une dépendance de y par rapport à x aux instants définis par l'horloge h.

$$G = \emptyset \mid G \cup \{f : (?\ \vec{x}\ !\ \vec{y})\ g\}$$

$$G \vdash x := y\$1 \Rightarrow (\widehat{x} = \widehat{y}) \quad G \vdash x := y \textbf{ default } z \Rightarrow \left(\begin{array}{c} y \xrightarrow{\widehat{y}} x, z \xrightarrow{\widehat{z}\backslash\widehat{y}} x \\ \widehat{x} = \widehat{y} \vee \widehat{z} \end{array} \right)$$
$$G \vdash x := y \textbf{ when } z \Rightarrow \left(\begin{array}{c} y \xrightarrow{\widehat{x}} x \\ \widehat{x} = \widehat{y} \wedge [z] \end{array} \right) \quad G \vdash x := o(\vec{y}) \Rightarrow \left(\begin{array}{c} y \xrightarrow{\widehat{y}} x \\ \widehat{x} = \widehat{y} \end{array} \right)_{y \in \vec{y}}$$
$$\frac{G \vdash p \Rightarrow g \quad G \vdash p' \Rightarrow g'}{G \vdash p|p' \Rightarrow g \cup g'} \qquad \frac{G \vdash p \Rightarrow g}{G \vdash p/x_1, \cdots, x_n \Rightarrow \forall x_1 \cdots \forall x_n.g}$$

$$\frac{G \vdash p' \Rightarrow g' \quad G \cup \{f : (?\ \vec{x}\ !\ \vec{y})\ g'\} \vdash p \Rightarrow g}{G \vdash p \textbf{ where process } f(?\ \vec{x}\ !\ \vec{y})\ p' \Rightarrow g} \qquad \frac{f : (?\ \vec{x}\ !\ \vec{y})\ g \in G}{G \vdash \vec{y}' := f(\vec{x}') \Rightarrow g[\vec{x}'/\vec{x}, \vec{y}'/\vec{y}]}$$

FIG. 6: *Le calcul d'horloges*

Certaines conditions sont à vérifier :

– Le graphe de dépendance ne doit pas contenir de cycles.

– Les relations d'horloge ne doivent ni forcer des horloges booléennes à être toujours vraies ou toujours fausses ni forcer la nullité de toutes les horloges (ce qui signifierait que le système ne réagit pas).

Dans le cas d'un retard ($x := y\$1$), les signaux x et y doivent être synchrones. Il n'y a pas de dépendance car la valeur de x n'est pas déterminée par la valeur actuelle de y mais par sa valeur précédente.

Dans le cas d'une extraction ($x := y$ **when** z), l'horloge de x est égale à l'intersection de l'horloge de y et de l'horloge à laquelle z est présent et vrai. x dépend de y à l'horloge $\widehat{y} \wedge [x]$.

Dans le cas d'un mélange ($x := y$ **default** z), l'horloge de x est égale à l'union des horloges de x et y. x dépend de y à l'horloge de y, de z à l'horloge $\widehat{z} \setminus \widehat{y}$.

Dans le cas d'une fonction instantanée ($x := o(y_1, \cdots, y_n)$), les signaux $x, y_1, \cdots, y_n$ sont synchrones. x dépend de chaque signal y_i à l'horloge $\widehat{x}$.

1.4 Le processus de compilation de SIGNAL

Après le contrôle des types et le calcul d'horloges, les horloges sont hiérarchisées : Un algorithme heuristique produit, à partir des relations d'horloge, une forêt d'horloges. Celle-ci est un ensemble d'arbres dont les racines sont les horloges les plus rapides et dont les nœuds ont pour fils les arbres des horloges extraites. Ce graphe hiérarchisé est alors combiné avec le GDC pour produire le Graphe Hiérarchisé aux Dépendances Conditionnées (GHDC) ([1], [3] et [8]). Chaque nœud du GHDC est une horloge à laquelle on associe les signaux calculables lorsque cette horloge est présente.

1.4.1 Schéma de compilation de SIGNAL

Montrons sur un exemple le schéma de compilation de SIGNAL . Le calcul d'horloge du processus $x := y$ **default** z introduit dans le GDC les arcs $y \xrightarrow{\widehat{y}} x$ et $z \xrightarrow{\widehat{z}\setminus\widehat{y}} x$. Le compilateur en infère le code séquentiel suivant :

```
if present(x) then
  if present(y) then
    x := y
```

```
    else
      x := z
    endif
  endif
```

1.4.2 Triangularisation des équations d'horloges

L'analyse d'un programme SIGNAL nous donne, suivant la fig. 6, un ensemble de relations d'horloges. Le compilateur va *triangulariser* ces relations, i.e. il va les transformer en une liste d'équations de la forme $h = h_1$ op h_2, où les opérandes h_1 et h_2 ont été définies avant h.

1.4.3 Représentation hiérarchique des équations

À partir du système triangularisé, Le compilateur SIGNAL construit une représentation hiérarchique des horloges (le graphe hiérarchisé). Les racines sont les horloges les plus rapides parce que ce sont soit des événements produits par l'environnement du système réactif (i.e. des horloges d'entrée), soit des horloges contraintes dont on n'a pas pu trouver une définition explicite en fonction des autres horloges. Chaque nœud a pour père l'horloge dont il est extrait.

En partant de cette représentation et à l'aide du schéma de compilation, il est aisé de produire un code séquentiel.

2 Typage de SIGNAL

Le système de types que nous proposons pour SIGNAL est une sémantique statique qui permet de détecter à la compilation des programmes qui pourraient ne pas s'exécuter correctement. Par exemple, le processus suivant

```
  y := x+1
| x := 1 when b default true
```

est syntaxiquement correct et est valable suivant la sémantique dynamique de SIGNAL . Mais il se termine par une erreur si b vaut `false` ou n'est pas défini. Alternativement, on pourrait dire d'un tel programme qu'il définit `y` entier à

l'horloge `b`. Malheureusement, combiner typage et horloge n'est pas toujours décidable statiquement (par exemple si `b` fait référence à une expression arithmétique).

Après formalisation du système de types de SIGNAL , nous pourrons démontrer le théorème d'auto-réduction qui assure qu'un processus bien typé se réduit dans la sémantique dynamique en un processus bien typé. Autrement dit, le contrôle statique des types garantit que le programme ne s'arrêtera pas sur une erreur de type.

2.1 Le polymorphisme

Dans la version présente de SIGNAL (la V4) il est possible de définir des processus génériques. Par exemple :

```
(| a := id(true)
 | b := id(1) |)/a,b
where
  process id = (?x !y)
    (| y := x |)
end
```

À chaque appel de ce processus, le compilateur substitue le code puis en vérifie le typage. L'exemple devient alors :

```
(| (y := x)[true/x, a/y]
 | (y := x)[1/x, b/y] |)
```

Mais si le processus `id` a été compilé séparément, la substitution n'est plus possible. L'introduction du polymorphisme en SIGNAL permet de résoudre ce problème. Le synthétiseur de type va donner à `id` le type polymorphe $\forall\alpha.\forall\alpha'.\alpha \rightarrow \alpha' \,|\, \alpha \leq \alpha'$ puis instancier α et α' à chaque appel de `id`. Nous avons vu au chapitre précédent que la compilation d'un processus SIGNAL consiste à le transformer en un GHDC. Or la construction d'un GHDC ne nécessite pas la connaissance des types. On pourra donc compiler séparément un processus polymorphe.

2.2 Les opérateurs surchargés

La *surcharge* [10] c'est l'utilisation d'un même symbole pour différents opérateurs. Seul le contexte permet de déterminer quel est l'opérateur référencé par le symbole. Par exemple le symbole + référence à la fois l'addition sur les entiers et celle sur les réels. Les opérateurs arithmétiques ainsi que les opérateurs de comparaison sont surchargés en SIGNAL . Le système de modules que nous proposerons au chapitre suivant permet de définir des opérateurs surchargés. Cette surcharge est résolue avec l'aide du programmeur : Soit l'opérateur sera pointé par son module correspondant, soit on choisira le dernier module ouvert définissant l'opérateur. Nous utiliserons donc les modules pour définir les opérateurs surchargés de SIGNAL .

2.3 Sous-typage et contraintes

L'ensemble des types de SIGNAL est doté d'une relation de *sous-typage*. Cette relation, notée $\leq$, est un ordre qui détermine quand un type peut en remplacer un autre. Un type t est un *sous-type* du type t' si là où est attendu un terme de type t', un terme de type t peut être utilisé. En particulier, un processus qui attend un signal d'entrée de type t', peut recevoir un signal de type t. La relation de sous-typage est réflexive, transitive et antisymétrique. Les types atomiques de SIGNAL vérifient les relations indiquées en fig. 7.

$$C \supset \textbf{event} \leq \textbf{boolean} \quad C \supset \textbf{short} \leq \textbf{integer} \quad C \supset \textbf{integer} \leq \textbf{long}$$

$$C \supset \textbf{real} \leq \textbf{dreal} \quad C \supset \textbf{complex} \leq \textbf{dcomplex}$$

$$C \supset t \leq t \qquad \frac{C \supset t \leq t' \quad C \supset t' \leq t''}{C \supset t \leq t''}$$

$$C \cup \{t \leq t'\} \supset t \leq t' \qquad \frac{C \supset t \leq t' \quad C \supset t' \leq t}{C \supset t = t'}$$

FIG. 7: *La relation de sous-typage*

Une *contrainte* est un élément de la relation de sous-typage. Nous introduisons dans la spécification du typage un ensemble C de contraintes. Par exemple, lors du typage de l'expression e **when** e' (cf. fig. 8), e' peut être soit de type **event** soit de type **boolean** . C doit donc contenir la contrainte $t' \leq$ **boolean**, t' étant le type de l'expression e'.

La notation $C \supset t \leq t'$, signifie que de C on peut déduire la contrainte $t \leq t'$. En d'autre termes, la fermeture transitive de C contient la contrainte $t \leq t'$.

2.4 Spécification du typage

Nous spécifions le système de types de SIGNAL sous forme de règles d'inférence précisant en haut les prémisses et en bas la conclusion.

2.4.1 Typage d'une expression sur signaux

Les règles de typage d'une expression sur signaux sont résumées en fig. 8.

$$\frac{\mathcal{R}(v) = t}{A, C \vdash_e v : t} \quad \frac{A(x) = t}{A, C \vdash_e x : t} \quad \frac{A, C \vdash_e v : t \quad A, C \vdash_e e : t}{A, C \vdash_e e\$1 \textbf{ init } v : t}$$

$$\frac{A, C \vdash_e e : t \quad A, C \vdash_e e' : \textbf{boolean}}{A, C \vdash_e e \textbf{ when } e' : t} \quad \frac{A, C \vdash_e e : t \quad A, C \vdash_e e' : t}{A, C \vdash_e e \textbf{ default } e' : t}$$

$$(\text{subsumption}) \frac{A, C \vdash_e e : t \quad C \supset t \leq t'}{A, C \vdash_e e : t'}$$

FIG. 8: *Spécification du typage d'une expression sur signaux*

A est l'environnement de typage. C'est un ensemble qui contient tous les signaux et les opérateurs visibles associés à leur type.

La règle de *subsumption* signifie que dans un environnement de typage A, si l'on peut typer l'expression e par t et que t est un sous-type de t', alors e est aussi de type t'.

$\mathcal{R}$ est la fonction qui associe à chaque constante son type minimal.

Prenons le cas du **when** : Pour typer e **when** e' par t dans A, il faut montrer que e est de type t, et que e' est de type **boolean** . Si e' est de type **event** , alors on applique la règle de subsumption.

2.4.2 Définition et appel de processus

Reprenons l'exemple :

```
(| a := id(true)
 | b := id(1) |)/a,b
where
  process id = (?x !y)
    (| y := x |)
end
```

Au typage de `id`, il n'est pas possible de déterminer les types de `x` et `y`. Des variables de type α et α' sont respectivement attribuées à `x` et `y` sous la contrainte $\alpha \leq \alpha'$ et `id` a alors le type $\alpha \to \alpha' \mid \alpha \leq \alpha'$. Ensuite, `id` étant appliqué à un booléen, on en déduit que $\alpha =$ **boolean** et la contrainte devient **boolean** $\leq \alpha'$. lorsque `id` est appliqué à l'entier 1 il y a une erreur de type. Ceci nous amène à introduire les notions de *généralisation* et d'*instanciation.*

La généralisation consiste à quantifier universellement toutes les variables libres n'apparaissant pas dans l'environnement de typage A. Un type t généralisé sera dit *schéma de type* et sera noté σ.

Par exemple, le schéma de type de `id` sera $\forall\alpha.\forall\alpha'.\alpha \to \alpha' \mid \alpha \leq \alpha'$.

A chaque appel (cf. fig. 9), le schéma de type du proce ssus doit être instancié, c'est-à-dire qu'une nouvelle variable de type est attribuée à chaque variable quantifiée et les quantificateurs sont supprimés.

$$\frac{(A, C \vdash_e e_i : t_i)_{i=1,\cdots,n} \quad t_1 \times \cdots \times t_n \to t'_1 \times \cdots \times t'_m / C \preceq A(f)}{A, C \vdash_e f(e_1, \cdots, e_n) : t'_1 \times \cdots \times t'_m}$$

FIG. 9: *Appel d'un processus*

Dans l'exemple, `id` est instancié deux fois. Les variables étant nouvelles à chaque instanciation, il n'y a pas d'erreur de type.

Définition 1 *Le symbole $\preceq$ signifie «est une instance de». $\tau/C \preceq \forall\alpha_1 \cdots \forall\alpha_n.\tau'/C'$ si et seulement si $\tau = \tau'[t_i/\alpha_i]_{i=1,\cdots,n}$ et $C \supset C'[t_i/\alpha_i]_{i=1,\cdots,n}$*

Définition 2 *σ est plus général que σ' si et seulement si pour tout $\tau/C, \tau/C \preceq \sigma \Rightarrow \tau/C \preceq \sigma'$.*

2.4.3 Typage d'un processus

Les règles de typage d'un processus sont résumées en fig. 10.

$$\frac{(A, C \vdash_e x_i : t_i)_{i=1,\cdots,n} \quad A, C \vdash_e e : t_1 \times \cdots \times t_n}{A, C \vdash_p x_1, \cdots, x_n := e}$$

$$\frac{A, C \vdash_p p \quad A, C \vdash_p p'}{A, C \vdash_p p|p'} \qquad \frac{A \cup \{x_1 : t_1, \cdots, x_n : t_n\}, C \vdash_p p}{A, C \vdash_p p/x_1, \cdots, x_n}$$

$$\frac{A, C \vdash_p p}{A, C \vdash_p p \textbf{ where } \emptyset \textbf{ end}} \qquad \frac{A \cup \{a : t\}, C \vdash_p p \textbf{ where } dl \textbf{ end}}{A, C \vdash_p p \textbf{ where } a; dl \textbf{ end}}$$

$$\frac{\begin{array}{c} A \cup \{a_1 : t_1, \cdots, a_n : t_n, a'_1 : t'_1, \cdots, a'_m : t'_m\}, C \vdash_p p' \\ A \cup \{f : gen(A, C, t_1 \times \cdots \times t_n \to t'_1 \times \cdots \times t'_m)\}, C \vdash_p p \textbf{ where } dl \textbf{ end} \end{array}}{A, C \vdash_p p \textbf{ where process } f = (?\ a_1, \cdots, a_n\ !\ a'_1, \cdots, a'_m)\ p'; dl \textbf{ end}}$$

FIG. 10: *Spécification du typage d'un processus*

Un processus n'a pas de type. Il peut simplement être bien typé.
Une équation est bien typée si :

- les signaux à gauche de l'opérateur $:=$ sont dans l'environnement de typage A (i.e. Ils ont été déclarés) et
- le type de l'expression e à droite de l'opérateur $:=$ est égal au type du n-uplet de signaux à gauche de $:=$.

Une composition de processus est bien typée si les processus composés sont bien typés.

Lors de la déclaration d'un processus local, ce processus est typé puis son identificateur f est ajouté à l'environnement de typage A affecté du type généralisé du processus. La généralisation $gen(A, C, \tau)$ est égale à $\forall\alpha_1.\cdots.\forall\alpha_n.\tau/\bar{C}_{\alpha_1,\cdots,\alpha_n}$ où $\{\alpha_1, \cdots, \alpha_n\}$ est l'ensemble des variables de τ et C qui ne sont pas libres dans A, et $\bar{C}_{\alpha_1,\cdots,\alpha_n}$ la fermeture transitive de C restreinte aux contraintes dont l'un des membres au moins contient une variable de $\{\alpha_1, \cdots, \alpha_n\}$.

Définition 3 *$gen(A, C, \tau) = \forall\alpha_1.\cdots.\forall\alpha_n.\tau/\bar{C}_{\alpha_1,\cdots,\alpha_n}$ si et seulement si $\{\alpha_1, \cdots, \alpha_n\} = (fv(\tau) \cup fv(C))\backslash fv(A)$*

2.4.4 Correction du typage vis-à-vis de la sémantique dynamique

Le théorème d'auto-réduction affirme qu'un processus p bien typé se réduit en un processus p' bien typé. Le typage est invariant au cours de l'exécution.

Définition 4 *$A, C \vdash m$ si et seulement si pour tout $x \in dom(m)$, $A, C \vdash_e m(x) : A(x)$.*

On note $m|_{input(p)}$ (resp. $m|_{output(p)}$) la restriction de m aux signaux d'entrée (resp. de sortie).

Théorème 1 *Pour tout environnement de typage A, tout ensemble de contraintes C, tout processus p (en syntaxe élémentaire), tout événement m, si $A, C \vdash_p p$, $A, C \vdash m|_{input(p)}$ et $p \xrightarrow{m} p'$ alors $A, C \vdash_p p'$ et $A, C \vdash m|_{output(p)}$.*

Preuve. Par induction sur la définition de $\rightarrow$:

1. Les cas de la fonction instantanée, de l'extraction (**when**) et du mélange (**default**) sont triviaux car l'état ne change pas.

2. Si $A, C \vdash_p x := y\$1$ **init** u alors on a la preuve de typage suivante :

$$\frac{A, C \vdash_e x : t \quad \dfrac{A, C \vdash_e y : t \quad A, C \vdash_e u : t}{A, C \vdash_e y\$1 \textbf{ init } u : t}}{A, C \vdash_q x := y\$1 \textbf{ init } u}$$

On a la transition suivante :

$$x := y\$1 \textbf{ init } u \xrightarrow{x(u)\ y(v)} x := y\$1 \textbf{ init } v$$

par hypothèse, $A, C \vdash \{x(v)\}$, donc $A \vdash_e v : t$. D'où on peut remplacer dans la preuve de typage, les occurrences de u par v.

3. dans le cas de la composition de processus, on a la transition :

$$\frac{p \xrightarrow{m} p'' \quad p' \xrightarrow{m'} p''' \quad m \cap m'}{p|p' \xrightarrow{m \cup m'} p''|p'''}$$

Par hypothèse $A, C \vdash_p p|p'$, on a donc la preuve de typage :

$$\frac{A, C \vdash_p p \quad A, C \vdash_p p'}{A, C \vdash_p p|p'}$$

Par l'hypothèse d'induction on a $A, C \vdash_p p''$ et $A, C \vdash_p p'''$. On peut donc remplacer dans la preuve de typage les occurrences de p (resp. p') par p'' (resp. p''').

□

2.5 Synthèse des types

Pour qu'un ensemble de règles d'inférence forme un algorithme déterministe, il doit vérifier les deux propriétés suivantes :

- Il ne doit pas y avoir d'ambiguïtés sur le choix de la règle à appliquer. Pour un terme donné, une seule règle doit être applicable.
- Les règles doivent vérifier la propriété de la sous-formule : Toutes les formules des prémisses doivent être des sous-formules de celles de la conclusion.

Nous sommes donc amenés à transformer notre ensemble de règles en un ensemble équivalent qui vérifie ces propriétés. Les règles modifiées sont décrites en figure 11.

$$\frac{\mathcal{R}(v) = t \quad C \supset t \leq \alpha}{A, C \vdash_e v : \alpha} \qquad \frac{A(x) = t}{A, C \vdash_e x : t}$$

$$\frac{A, C \vdash_e v : t \quad A, C \vdash_e e : t' \quad C \supset t \leq \alpha, t' \leq \alpha}{A, C \vdash_e e\$1 \textbf{ init } v : \alpha}$$

$$\frac{A, C \vdash_e e : t \quad A, C \vdash_e e' : t' \quad C \supset t \leq \alpha, t' \leq \textbf{boolean}}{A, C \vdash_e e \textbf{ when } e' : \alpha}$$

$$\frac{A, C \vdash_e e : t \quad A, C \vdash_e e' : t' \quad C \supset t \leq \alpha, t' \leq \alpha}{A, C \vdash_e e \textbf{ default } e' : \alpha}$$

$$\frac{\begin{array}{cc}(A, C \vdash_e e_i : t_i)_{i=1,\cdots,n} & (C \supset t_i \leq t'_i)_{i=1,\cdots,n} \\ t'_1 \times \cdots \times t'_n \to t''_1 \times \cdots \times t''_m / C \preceq A(f) & (C \supset t''_i \leq \alpha_i)_{i=1,\cdots,m}\end{array}}{A, C \vdash_e f(e_1, \cdots, e_n) : \alpha_1 \times \cdots \times \alpha_m}$$

FIG. 11: *Algorithme de typage*

Si les règles d'inférence vérifient ces propriétés, alors on en déduit un programme fonctionnel dont le squelette est un filtrage (*pattern matching*). Or ici, quelque soit l'expression on peut toujours appliquer la règle de *subsumption*. Plutôt que d'appliquer la règle de subsumption durant le déroulement, nous accumulons les relations de sous-typage dans un ensemble global C. Par exemple, lors du typage de l'expression e **when** e', e' peut être soit de type **event** soit de type **boolean** . Nous ajoutons alors à C la contrainte $t' \leq$ **boolean**, t' étant le résultat de l'appel récursif de l'algorithme de typage sur l'expression e'.

On remarque que les contraintes sont basiques : Elles ne font intervenir que des types de base ou des variables de types. Ces contraintes sont accumulées durant le déroulement de l'algorithme de typage. Leur cohérence sera vérifiée après le déroulement de l'algorithme de typage. On ne peut pas vérifier les contraintes dès leur introduction dans C car elles peuvent faire intervenir des variables de types qui seront résolues ultérieurement. Si la fermeture

transitive de l'ensemble C des contraintes contient une contradiction (tel que *integer* $\leq$ *short*), alors il y a une erreur de typage dans le processus.

2.5.1 Correction de l'algorithme vis-à-vis de la sémantique statique.

Il faut vérifier que l'algorithme de synthèse de type respecte la spécification du typage. Pour cela nous allons montrer deux théorèmes :

- Le théorème de *cohérence* qui dit que si dans un environnement A, l'algorithme type e avec le type t et les contraintes C, alors c'est prouvable à l'aide de la spécification.
- Le théorème de *complétude* qui dit que tout programme typable suivant la spécification l'est aussi par l'algorithme.

Théorème 2 (cohérence) *Si l'algorithme de typage donne $A, C \vdash e : t$ alors on peut prouver $A, C \vdash e : t$ avec la spécification du typage.*

Preuve. Il suffit de remplacer dans la preuve de typage par l'algorithme :

- les occurrences de :
$$\frac{\mathcal{R}(v) = t \quad C \supset t \leq \alpha}{A, C \vdash_e v : \alpha}$$
par :
$$\frac{\dfrac{\mathcal{R}(v) = t}{A, C \vdash_e c : t} \quad C \supset t \leq \alpha}{A, C \vdash_e c : \alpha}$$
- les occurrences de :
$$\frac{\mathcal{R}(v) = t \quad A, C \vdash_e e : t' \quad C \supset t \leq \alpha, t' \leq \alpha}{A, C \vdash_e e\$1 \textbf{ init } v : \alpha}$$
par :
$$\frac{\dfrac{\dfrac{\mathcal{R}(v) = t}{A, C \vdash_e v : t} \quad C \supset t \leq \alpha}{A, C \vdash_e v : \alpha} \qquad \dfrac{A, C \vdash_e e : t' \quad C \supset t' \leq \alpha}{A, C \vdash_e e : \alpha}}{A, C \vdash_e e\$1 \textbf{ init } v : \alpha}$$

– les occurrences de :

$$\frac{A, C \vdash_e e : t \quad A, C \vdash_e e' : t' \quad C \supset t \leq \alpha, t' \leq \mathbf{boolean}}{A, C \vdash_e e \ \mathbf{when}\ e' : \alpha}$$

par :

$$\frac{\dfrac{A, C \vdash_e e : t \quad \dfrac{A, C \vdash_e e' : t' \quad C \supset t' \leq \mathbf{boolean}}{A, C \vdash_e e' : \mathbf{boolean}}}{A, C \vdash_e e \ \mathbf{when}\ e' : t} \quad C \supset t \leq \alpha}{A, C \vdash_e e \ \mathbf{when}\ e' : \alpha}$$

– les occurrences de :

$$\frac{A, C \vdash_e e : t \quad A, C \vdash_e e' : t' \quad C \supset t \leq \alpha, t' \leq \alpha}{A, C \vdash_e e \ \mathbf{default}\ e' : \alpha}$$

par :

$$\frac{\dfrac{A, C \vdash_e e : t \quad C \supset t \leq \alpha}{A, C \vdash_e e : \alpha} \quad \dfrac{A, C \vdash_e e' : t' \quad C \supset t' \leq \alpha}{A, C \vdash_e e' : \alpha}}{A, C \vdash_e e \ \mathbf{default}\ e' : \alpha}$$

– les occurrences de :

$$\frac{\begin{array}{cc}(A, C \vdash_e e_i : t_i)_{i=1,\cdots,n} & (C \supset t_i \leq t'_i)_{i=1,\cdots,n} \\ t'_1 \times \cdots \times t'_n \to t''_1 \times \cdots \times t''_m / C \preceq A(f) & (C \supset t''_i \leq \alpha_i)_{i=1,\cdots,m}\end{array}}{A, C \vdash_e f(e_1, \cdots, e_n) : \alpha_1 \times \cdots \times \alpha_m}$$

par :

$$\frac{\begin{array}{c}\left(\dfrac{A, C \vdash_e e_i : t_i \quad C \supset t_i \leq t'_i}{A, C \vdash_e e_i : t'_i}\right)_{i=1,\cdots,n} \\ \dfrac{t'_1 \times \cdots \times t'_n \to t''_1 \times \cdots \times t''_m / C \preceq A(f)}{A, C \vdash_e f(e_1, \cdots, e_n) : t''_1 \times \cdots \times t''_m} \quad \dfrac{(C \supset t''_i \leq \alpha_i)_{i=1,\cdots,m}}{t''_1 \times \cdots \times t''m \leq \alpha_1 \times \cdots \times \alpha_m}\end{array}}{f(e_1, \cdots, e_n) : \alpha_1 \times \cdots \times \alpha_m}$$

□

Définition 5 (Fermeture)

$$C \downarrow_A = \bigcup_{\alpha \in A} C \downarrow_\alpha \quad \textit{avec } C \downarrow_\alpha = C_\alpha \cup C \downarrow_{fv(C_\alpha) \setminus \{\alpha\}} \quad \textit{et } C_\alpha = \{\alpha \leq t, t \leq \alpha \in C\}$$

Théorème 3 (complétude) *Si* $\sigma(A), C \vdash p : \tau$ *avec* $\sigma(C) \subseteq C$ *alors*

- $A, C' \vdash p \Rightarrow \tau'$ *par l'algorithme*
- $\exists \sigma'$ *tel que* $\tau = \sigma'(\tau') \quad C \supseteq \sigma'(C' \downarrow_{fv(\tau) \cup fv(A)}) \quad \sigma(A) = \sigma'(A)$

Preuve. Par induction sur la preuve de $A, C \vdash p : \tau$ dans la spécification du typage. Voyons par exemple le cas du **when** :
Supposons qu'on ait les preuves Π_1 et Π_2 telles que :

$$\frac{(\Pi_1)\dfrac{\vdots}{\sigma(A), C \vdash_e e : t} \quad (\Pi_2)\dfrac{\vdots}{\sigma(A), C \vdash_e e' : \mathbf{boolean}}}{\sigma(A), C \vdash_e e \ \mathbf{when}\ e' : t}$$

Les hypothèses d'induction sont les suivantes :

- $A, C' \vdash_e e \Rightarrow t'$ par l'algorithme
- $\exists \sigma'$ tel que $t = \sigma'(t') \quad C \supseteq \sigma'(C') \quad \sigma(A) = \sigma'(A)$

et

- $A, C' \vdash_e e' \Rightarrow t''$ par l'algorithme
- $\exists \sigma''$ tel que $\mathbf{boolean} = \sigma''(t'') \quad C \supseteq \sigma''(C') \quad \sigma(A) = \sigma''(A)$

Par les hypothèses d'induction et la règle de typage du **when** dans l'algorithme, on en déduit :

$A, C' \vdash_e e$ when $e' \Rightarrow \alpha$ avec α variable fraiche et $C' \supset t'' \leq \mathbf{boolean}, t \leq \alpha$

Il ne reste plus qu'à construire σ''' par

- $\sigma''' = \sigma'$ pour les variables libres de A et t
- $\sigma''' = \sigma''$ pour les variables libres de t''
- $\sigma'''(\alpha) = t$

On a bien $\sigma'''(\alpha) = t$ et $\sigma(A) = \sigma'''(A)$.
On a :

- $C \supseteq \sigma'(C') \downarrow_{fv(A) \cup fv(\alpha)}$ pour les contraintes sur les variables de A et t
- $C \supseteq \sigma''(C') \downarrow_{fv(A) \cup fv(\alpha)}$ pour les contraintes sur les variables de t'
- $C \supseteq \sigma'''(t \leq \alpha)$ pour α

d'où $C \supseteq \sigma'''(C')$ par définition de σ''' □

2.6 simplification des contraintes

Soit C l'ensemble brut des contraintes obtenues lors de l'application de l'algorithme de typage à un processus f. Soit T l'ensemble des variables de type apparaissant dans l'interface de ce processus f. Nous commençons la simplification en calculant la restriction C_T de C en supprimant les contraintes dont l'un des membres contient une variable α n'appartenant pas à l'ensemble A des variables d'interface.

$$C_T = C \cup (\bigcup_{\alpha \notin T} \{\tau \leq \tau' | \tau \leq \alpha \in C, \alpha \leq \tau' \in C\}) \backslash (\bigcup_{\alpha \notin T} \{\tau \leq \alpha, \alpha \leq \tau'\})$$

Nous considérons l'ensemble des types de base ordonnés par la relation de sous-typage. Nous appliquons à C_T l'algorithme défini en fig. 12. Soit C' le résultat.

$C_T \rightarrow C'$

$$\frac{t_1 \leq t_2 \quad C \rightarrow C'}{\{t_1 \leq t_2\} \cup C \rightarrow C'} \qquad \frac{t \text{ minima} \quad C[t|\alpha] \rightarrow C'}{\{\alpha \leq t\} \cup C \rightarrow C'} \qquad \frac{t \text{ maxima} \quad C[t|\alpha] \rightarrow C'}{\{t \leq \alpha\} \cup C \rightarrow C'}$$

FIG. 12: *Simplification des contraintes*

Théorème 4 *Pour tout t et t' tels que t (resp. t') soit un type de base ou une variable appartenant à A : $C \supset t \leq t' \Leftrightarrow C' \supset t \leq t'$*

3 Système de modules

Nous allons définir un système de modules ([5], [6] et [7]) pour SIGNAL permettant de définir des unités génériques, des types abstraits et de paramétrer les modules par d'autres modules. Nous montrerons que dans le cas de SIGNAL , nous pouvons identifier module et unité de compilation (au sens de SIGNAL) sans perdre d'expressivité. Le découpage en modules permet de structurer le code afin d'en améliorer la compréhension : Un module peut être compris séparément. Le découpage en unités de compilation, permet de compiler séparément différentes parties d'un programme, ce qui permet de minimiser l'utilisation des ressources en mémoire et en temps de l'ordinateur.

3.1 Le langage de modules

Le système de modules est un langage indépendant de Signal (cf. fig. 13). Les valeurs manipulées par ce langage sont les modules.

$$
\begin{array}{rcl}
m & ::= & \textbf{type } T = t \\
 & | & \textbf{constant } V = v \\
 & | & d \\
 & | & m; m'
\end{array}
\qquad
\begin{array}{rcl}
s & ::= & \textbf{type } T \\
 & | & \textbf{type } T = t \\
 & | & \textbf{constant } V : t \\
 & | & \textbf{process } f(?\ \vec{x} : \vec{t}\ !\ \vec{y} : \vec{t'})/C\ \textbf{spec } g \\
 & | & s; s'
\end{array}
$$

$$
\begin{array}{rcl}
P & ::= & \textbf{signature } S = s \textbf{ end} \\
 & | & \textbf{functor } F(M_1 : S_1, \cdots, M_n : S_n) \\
 & | & \textbf{module } M = F(M_1, \cdots, M_n) \\
 & | & \textbf{module } M = m \textbf{ end} \\
 & | & P; P'
\end{array}
$$

FIG. 13: *Le langage de modules*

Un module est un ensemble de définitions de types, de constantes et de processus. Un *foncteur* est un module paramétré par d'autres modules. La notion de foncteur apporte la possibilité de définir des modules génériques.

Ces valeurs (les modules) sont typées. Le type d'un module est une *signature*. Une signature est un ensembles de déclarations de types (abstraits ou pas), de constantes (avec leur type) et de processus (avec leur interface comprenant les types des signaux d'interface et les dépendances et relations d'horloges des signaux d'interface).

3.2 Sous-typage entre signatures

Soit un foncteur F attendant comme paramétre un module de signature S. Si M a pour pour signature S' et que S' est une sous-signature de S (on notera $S' \leq S$), alors le module M peut être passé comme paramétre au foncteur F.

Les différents cas de sous typage sont l'oubli d'une composante, l'oubli d'un égalité de type, une permutation de composantes, ou le remplacement d'une composante processus par un processus dont le type est sous-type du type attendu. Le sous-typage entre signatures est formalisé en fig. 14.

$$\frac{\rho : \{1, \cdots, n\} \longrightarrow \{1, \cdots, m\} \text{ injection} \quad (A \cup s \vdash s[\rho(i)] \leq s[i])_{i=1,\cdots,n}}{A \vdash s \leq s'}$$

$$A \vdash \textbf{type } T \leq \textbf{type } T \quad A \vdash \textbf{type } T = t \leq \textbf{type } T$$

$$\frac{A \vdash t \approx t'}{A \vdash \textbf{type } T = t \leq \textbf{type } T = t'} \quad \frac{A \vdash t \approx t'}{A \vdash \textbf{constant } V : t \leq \textbf{constant} V : t'}$$

$$\frac{\begin{array}{c} t''_1 \times \cdots \times t''n \to t'''_1 \times \cdots \times t'''_m \preceq gen(A, C, t_1 \times \cdots \times t_n \to t'_1 \times \cdots \times t'_m) \\ g \leq g'[x_1, \cdots, x_n, y_1, \cdots, y_m / x'_1, \cdots, x'_n, y'_1, \cdots, y'_m] \end{array}}{\begin{array}{c} A \vdash \textbf{process} f(?\ x_1 : t_1, \cdots, x_n : t_n\ !\ y_1 : t'_1, \cdots, y_m : t'_m)/C \textbf{ spec } g \leq \\ \textbf{process} f(?\ x'_1 : t''_1, \cdots, x'_n : t''_n\ !\ y'_1 : t'''_1, \cdots, y'_m : t'''_m)/C \textbf{ spec } g' \end{array}}$$

FIG. 14: *Sous-typage entre signatures*

3.3 Sous-typage entre spécifications de processus

$g \leq g'$ signifie que là où un processus de spécification g' est attendu, il peut y avoir un processus de spécification g. Le remplacement de $\vec{g}'$ par $\vec{g}$ ne doit jamais créer de cycles; si $\vec{g}$ avait des arcs supplémentaires, cela pourrait créer un cycle. En revanche, $\vec{g}$ peut avoir des arcs en moins. Il faut donc que $\vec{g} \subseteq \vec{g}'$.

Le remplacement de $\hat{g}'$ par $\hat{g}$ ne doit ni contraindre des horloges booléennes à être toujours vraies ou toujours fausses, ni contraindre la nullité de toutes les horloges (ce qui signifierait que le système ne réagit pas), ni créer de cycle dans le graphe de dépendance. Cela pourrait se produire si $\hat{g}$ amenait de nouvelles relations d'horloge. Il faut donc que $\hat{g} \Leftarrow \hat{g}'$ (cf. fig. 15), ce qui signifie que l'ensemble des solutions de $\hat{g}'$ est inclus dans celui de $\hat{g}$.

$$\frac{\vec{g} \subseteq \vec{g'} \quad \hat{g} \Leftarrow \hat{g}'}{g \leq g'}$$

FIG. 15: *Sous-typage entre spécifications de processus*

$\hat{g}$ et $\hat{g}'$ sont deux systèmes d'équations booléennes que l'on peut exprimer dans $\mathbb{Z}/2\mathbb{Z}$ ([1]) muni de ses opérations habituelles $+$ et $.$:

$$H : (P_i(X) = 0)_{i \in I} \qquad H' : (P'_j(X) = 0)_{j \in J}$$

où les P_i et les P'_j sont des polynômes de $\mathbb{Z}/2\mathbb{Z}$, I et J sont des ensembles d'indices, et X est le vecteur des variables du système d'équations.

Soit l'opérateur $\oplus$ défini par :

$$a \oplus b = a + b + a.b$$

Il est immédiat de vérifier que :

$$\begin{cases} P_1(X) = 0 \\ P_2(X) = 0 \end{cases} \Leftrightarrow (P_1 \oplus P_2)(X) = 0$$

On peut donc ramener chaque système d'équations à une seule équation.

$$H : P(X) = 0 \qquad H' : P'(X) = 0$$

où $P = \bigoplus_{i \in I} P_i$ et $P' = \bigoplus_{j \in J} P_j$.

Soient V et V' les ensembles de solutions de P et P'. La propriété suivante est immédiate :

propriété 1 *$V \subseteq V'$ si et seulement si* $\forall X(1 - P(X)).P'(X) = 0$

Vérifier que $\forall X(1-P(X)).P'(X) = 0$ revient donc à prouver que la formule booléenne correspondante est une tautologie.

3.4 Typage

3.4.1 Équivalence de types

La définition de types dans un module entraîne que le nouveau type est équivalent au type le définissant (cf. fig. 16). Cette relation d'équivalence (notée $\approx$) est utilisée pour le typage des processus.

$$\frac{A(T) = t}{A \vdash T \approx t}$$

FIG. 16: *Equivalence de types*

3.4.2 Ajouts au typage et au calcul d'horloge de SIGNAL

La notion de module nous amène à ajouter des règles d'inférence pour le typage des processus (cf. fig. 17). Il faut d'abord prendre en compte la relation d'équivalence de types introduite par la définition de types dans un module. Il faut aussi traiter les cas d'appel à un processus ou de référence à une constante ou un type définis dans un autre module M.

3.4.3 Typage d'un module

La spécification du typage d'un module est décrite en figure 18. On en déduit un algorithme de typage des modules en remontant la règle de *subsumption* dans chaque règle. Il n'est pas nécessaire d'introduire des contraintes car ici toutes les signatures (les types des modules) sont déclarées.

$$\frac{A, C \vdash e : t \quad A \vdash t \approx t'}{A, C \vdash e : t'} \qquad \frac{A \vdash M : s; \mathbf{type}\ T = t; s'}{A \vdash M.T \approx t[M.x/x]_{x \in s}}$$

$$\frac{A \vdash M : s; \mathbf{constant}\ V : t; s'}{A \vdash M.V : t[M.x/x]_{x \in s}}$$

$$\frac{A \vdash s; \mathbf{process}\ f(?\ \vec{x} : \vec{t}\ !\ \vec{y} : \vec{t'})/C'\ \mathbf{spec}\ g; s'}{A, C \vdash M.f : gen(A, C \cup C', \vec{t}[M.x/x]_{x \in s} \to \vec{t'}[M.x/x]_{x \in s})}$$

$$\frac{\mathbf{process}\ f(?\ \vec{x} : \vec{t}\ !\ \vec{y} : \vec{t'})/C\ \mathbf{spec}\ g \in A(M)}{G \vdash \vec{y'} := M.f(\vec{x'}) \Rightarrow g[\vec{x'}/\vec{x}, \vec{y'}/\vec{y}]}$$

FIG. 17: *Ajouts au typage et au calcul d'horloge de* SIGNAL

Le typage d'un module consiste à mettre les paramétres M_i avec leurs signatures s_i dans l'environnement de typage A puis à parcourir récursivement la liste m des composantes. Chaque composante cm_i, une fois traitée, est mise dans l'environnement de typage A car elle peut être utilisée par une composante ultérieure : Un processus peut appeler des processus ou utiliser des constantes définis avant lui. Les définitions de types sont mises dans l'environnement de typage A afin de pouvoir déterminer les équivalences de types lors du typage d'un processus.

Lors du typage de la définition d'un processus, on fait appel à l'algorithme de typage d'un processus SIGNAL défini au chapitre précédent. Il faut vérifier que les spécifications g de graphe de dépendance et de relations d'horloges correspondent bien au processus défini. Enfin on met dans l'environnement de typage A l'identificateur f du processus affecté du type généralisé du processus défini.

$$A \vdash \textbf{signature}\ S = s\ \textbf{end} : A \cup \{S : s\}$$

$$A \cup \{M : s\} \vdash M : s \quad A \cup \{S : s\} \vdash S : s$$

$$\frac{A(F) = (M'_1 : s_1 \times \cdots \times M'_n : s_n) \to s \quad (A \vdash M_i : s_i)_{i=1,\cdots,n}}{A \vdash \textbf{module}\ M = F(M_1, \cdots, M_n) : A \cup \{M : s[M_i/M'_i]_{i=1,\cdots,n}\}}$$

$$\frac{A \vdash m : s}{A \vdash \textbf{module}\ M = m\ \textbf{end} : A \cup \{M : s\}}$$

$$\frac{(A \vdash S_i : s_i)_{i=1,\cdots,n} \quad A \cup \{M_1 : s_1, \cdots, M_n : s_n\} \vdash m : s}{\begin{array}{c} A \vdash \textbf{functor}\ F(M_1 : S_1, \cdots, M_n : S_n) = m\ \textbf{end} : \\ A \cup \{F : (M_1 : s_1 \times \cdots \times M_n : s_n) \to s\} \end{array}}$$

$$\frac{A \cup \{T : t\} \vdash m : s}{A \vdash \textbf{type}\ T = t; m : \textbf{type}\ T = t; s} \qquad \frac{A \vdash P : A' \quad A' \vdash P' : A''}{A \vdash P; P' : A''}$$

$$\frac{A \vdash_e v : t \quad A \cup \{V : t\} \vdash m : s}{A \vdash \textbf{constant}\ V = v; m : \textbf{constant}\ V : t; s} \qquad \frac{A \vdash m : s' \quad A \vdash s' \leq s}{A \vdash m' : s}$$

$$\frac{\begin{array}{c} A \cup \{\vec{x} : \vec{t}, \vec{y} : \vec{t'}\}, C \vdash p \quad p \rightsquigarrow p' \quad G \vdash p' \Rightarrow g \\ A \cup \{f : gen(A, C, \vec{t} \to \vec{t'})\}, G \cup \{f : (?\ \vec{x}\ !\ \vec{y})\ g\} \vdash m : s \end{array}}{A \vdash \textbf{process}\ f(?\ \vec{x} : \vec{t}\ !\ \vec{y} : \vec{t'})\ p; m : \textbf{process}\ f(?\ \vec{x} : \vec{t}\ !\ \vec{y} : \vec{t'})/C\ \textbf{spec}\ g; s}$$

FIG. 18: *Vérification de typage*

3.5 Exemple d'utilisation du système de modules

Soit un foncteur `rectangle`(cf. fig 19), paramétré par un module `distance`. Un rectangle est défini par deux points `p1` et `p2` qui sont des sommets opposés du rectangle.

```
functor rectangle (distance : Distance) =
  type rectangle = struct (distance.point p1,p2)
  process perimetre = (? rectangle r ! real p)
    (| p1x := distance.get_x (p1)
     | p2y := distance.get_y (p2)
     | p3 := distance.create (p1x,p2y)
     | long1 := distance.distance (p1,p3)
     | long2 := distance.distance (p2,p3)
     | perimetre := 2*long1+2*long2
     |)/p1x,p2y,p3,long1,long2
  process surface = (? rectangle r ! real s)
    (| p1x := distance.get_x (p1)
     | p2y := distance.get_y (p2)
     | p3 := distance.create (p1x,p2y)
     | long1 := distance.distance (p1,p3)
     | long2 := distance.distance (p2,p3)
     | s := long1 * long
     |)/p1x,p2y,p3,long1,long2
end
```

FIG. 19: *Foncteur* `rectangle`

Ce module fournit deux processus `perimetre` et `surface` prenant en entrée un rectangle et renvoyant respectivement le périmètre et la surface du rectangle.

Les seules contraintes sur le module `distance` sont :

- il doit exporter un type `point`
- il doit fournir au moins les processus :
  - `create` pour créer un point
  - `get_x` et `get_y` pour récupérer l'abscisse et l'ordonnée d'un point

– `distance` pour calculer la distance entre deux points.

On impose donc que `distance` doit correspondre à la signature `Distance` (cf. fig 20).

```
signature Distance =
  type point
  process create = (?real x,y !point p)
    spec (| x^=y^=p | {x,y} --> p |)
  process get_x (?point p !real x) =
    spec (| p ^= x | p --> x |)
  process get_y (?point p !real y) =
    spec (| p ^= y | p --> y |)
  process distance = (?point A,B !real d)
    spec (| A ^= B ^= d | {A,B} --> d |)
end
```

FIG. 20: *Signature* `Distance`

Les signatures inférées des modules `distance1` et `distance2` (cf. fig. 21 et fig. 22) sont des sous-signatures de `Distance`. Donc, par application de la règle de subsumption, `distance1` et `distance2` admettent `Distance` comme signature.

Il ne reste plus qu'à appliquer le foncteur `rectangle` à `distance1` et `distance2` pour obtenir deux mises en œuvre.

```
module rectangle1=rectangle(distance1)
module rectangle2=rectangle(distance2)
```

3.6 Compilation d'un module

La compilation d'un module consiste a transformer chaque processus défini dans ce module en GHDC. Ce n'est qu'après l'édition de liens que du code (C, Fortran ou VHDL) sera généré. Or la construction d'un GHDC ne nécessite pas la connaissance des types. Les foncteurs sont donc des unités de compilation.

```
module distance1 =
  import math
  type point = struct (real x,y)
  process create = (?real x,y !point p)
    (| p := (x,y) |)
  process get_x (?point p !real x) =
    (| x := p.x |)
  process get_y (?point p !real y) =
    (| y := p.y |)
  process carre = (?real x !real y)
    (| y := x*x |)
  process distance = (?point a,b !real d)
    (| x := (a.x-a.x)
     | y := (a.y-a.y)
     | d := sqrt (carre(x)+carre(y)) |)/x,y
end
```

FIG. 21: *Module* `distance1`

Conclusion

Nous avons conçu et prototypé en système de modules avancé pour le langage SIGNAL. Il permet de définir des unités génériques, des types abstraits et de paramétrer les modules par d'autres modules.

Nous avons montré que dans le cas de SIGNAL, du fait de son mécanisme de compilation très particulier, on peut identifier modules et unités de compilation sans réduire l'expressivité des modules.

Le système de modules que nous proposons s'intègre dans l'environnement de SIGNAL. Il correspond à la méthodologie de programmation par raffinements successifs de spécifications grâce à ses mécanismes de types abstraits et de sous-typage des signatures de modules. Le prototype réalisé traduit un programme SIGNAL avec modules en un programme SIGNAL V4. Il pourra être intégré facilement dans le compilateur et servir de support à la compilation séparée.

```
module distance2 =
  type point = struct (real x,y)
  process create = (?real x,y !point p)
    (| p := (x,y) |)
  process get_x (?point p !real x) =
    (| x := p.x |)
  process get_y (?point p !real y) =
    (| y := p.y |)
  process distance = (?a,b !d)
    (| d := abs(a.x-b.x)+abs(a.y-b.y) |)
end
```

FIG. 22: *Module* `distance2`

Notre système de module permettra aussi d'interfacer des programmes SIGNAL avec des librairies externes.

Notre première tâche a été de décrire formellement le typage de Signal. Le système de type que nous proposons est une sémantique statique qui permet d'éliminer à la compilation des programmes qui, bien qu'ils soient acceptables par la sémantique dynamique, peuvent mener à une erreur. En usant des principes avancés de la théorie des types, nous avons déduit de cette spécification formelle un algorithme de synthèse automatique des types dont nous avons mis en œuvre un prototype.

Par la suite, nous pourrons étendre le système de types à l'aide des techniques de l'interprétation abstraite : Nous chercherons à combiner le typage avec le calcul d'horloge. Puis nous pourrons étudier les liens avec la modularisation des preuves de propriétés de programmes, la compilation séparée et la distribution des processus.

Références

[1] Tochéou Pascalin Amagbegnon. *Forme canonique arborescente des hor-*

loges de SIGNAL. PhD thesis, Université de Rennes 1, France, November 1995. in french - A paraître-.

[2] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[3] Loïc Besnard. *Compilation de* SIGNAL*: horloges, dépendances, environnement.* PhD thesis, Université de Rennes 1, France, September 1992. in french.

[4] Thierry Gautier, Paul Le Guernic, and François Dupont. SIGNAL v4 : manuel de référence (version préliminaire). Publication interne 832, IRISA, June 1994.

[5] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st symp. Principles of Programming Languages*, pages 109–122. ACM press, 1994.

[6] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd symp. Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[7] Xavier Leroy. Le système caml special light: modules et compilation efficace en caml. Rapport de recherche 2721, INRIA, November 1995.

[8] O. Maffeïs. *Ordonnancements de graphes de flots synchrones; Application à* SIGNAL. PhD thesis, Université de Rennes 1, France, January 1993. in french.

[9] David Nowak, Jean-Pierre Talpin, Thierry Gautier, and Paul Le Guernic. An ML-like module system for the synchronous language SIGNAL. In *Proceedings of European Conference on Parallel Processing (Euro-Par'97).* Springer Verlag LNCS, 1997. To appear.

[10] François Rouaix. Safe run-time overloading. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 355–366, 1990.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399